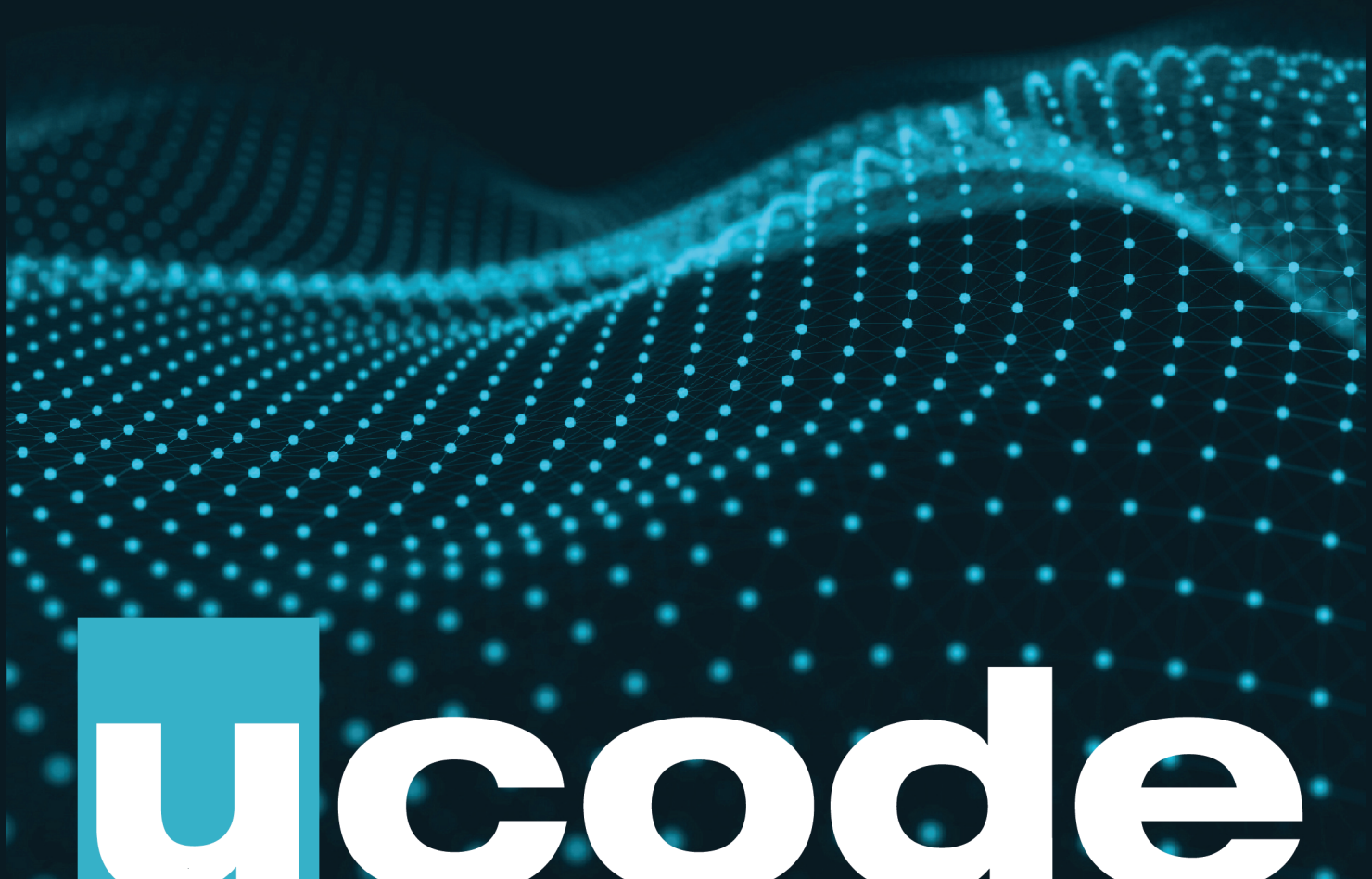
Sprint 06

Half Marathon C++

August 26, 2020



ucode

Contents

ucode

Engage

DESCRIPTION

Blessings of the moons upon you, traveler.

The key to success is daily self-improvement. If you want to learn how to use OOP to create a software architecture, you need to spend a lot of time on it. There is a lot to learn. Today we will continue to delve into the object-oriented paradigm.

So, in C++ there are terms like abstract class and interface. Both have a certain relation to inheritance, more precisely to modeling. They help to express that a certain group of things has something in common: general behavior, characteristics, and so on. Often an abstract class describes a model, and an interface describes a behavior or role. All of these elements greatly optimize development.

Let's get started!

BIG IDEA

Object-oriented programming.

ESSENTIAL QUESTION

What are the benefits of using OOP in software architecture?

CHALLENGE

Learn advanced OOP features of C++.

Investigate

GUIDING QUESTIONS

We invite you to find answers to the following questions. By researching and answering them, you will gain the knowledge necessary to complete the challenge. To find answers, ask the students around you and search the internet. We encourage you to ask as many questions as possible. Note down your findings and discuss them with your peers.

- What is an `interface`?
- What is an `abstract class`?
- In what situations should you use an interface instead of an abstract class?
- Why do we need an interface if we already have an abstract class?
- How to instantiate an abstract class?
- Why does an abstract class need a constructor?
- How do interfaces help to build a good architecture?
- What is the best use case abstract classes in system design?
- What is `operator overload` and how does this magic work?
- Why do we need operator overloading in C++?
- What is a `virtual method` in C++?

GUIDING ACTIVITIES

Complete the following activities. Don't forget that you have a limited time to overcome the challenge. Use it wisely. Distribute tasks correctly.

- Read the tasks carefully and try to find as much information as possible about them.
- Consider the algorithms found in the tasks.
- Allocate your resources and time.
- Use your research to carry out the tasks below.
- Discuss the tasks with students.
- Clone your git repository that is issued on the challenge page in the LMS.
- Try to implement your thoughts in code.
- Push the solution to the repository.

ANALYSIS

Analyze your findings. What conclusions have you made after completing guiding questions and activities? In addition to your thoughts and conclusions, here are some more analysis results.

- Be attentive to all statements of the story.
- Analyze all information you have collected during the preparation stages. Try to define the order of your actions.
- Perform only those tasks that are given in this document.

- Submit your files using the layout described in the story. Only useful files allowed, garbage shall not pass!
- Tasks in `shell` must be executed with `zsh`.
- Compile files with commands `cmake . -Bbuild && cmake --build ./build` that will call `CMake` and build an app.
- Pay attention to what is allowed in a certain task. Use of forbidden stuff is considered a cheat and your tasks will be failed.
- Complete tasks according to the rules specified in the Google C++ Style Guide.
  But there are several exceptions for the guide listed below:
  - you can use `#pragma once` directive instead of `#ifndef ... #define`
  - variables can be written in `mixed case`
  - class data members must begin with `m_` prefix (m for member)
  - indent 4 spaces at a time
  - each line of text in your code must be at most 120 characters long
  - ignore the sections Inputs and Outputs and Legal Notice and Author Line in the style guide
- The solution will be checked and graded by students like you. Peer-to-Peer learning.
- You must use this project structure for each task:

```
>tree project_dir --dirsfirst --charset=ascii
project_dir
|-- app
|   |-- src
|   |   |-- CMakeLists.txt
|   |   `-- ...
|   |-- resources
|   |   `-- ...
|   |-- CMakeLists.txt
|   `-- main.cpp
|-- lib1
|   |-- CMakeLists.txt
|   `-- ...
...
|-- libN
|   |-- CMakeLists.txt
|   `-- ...
`-- CMakeLists.txt
```

- If you have any questions or don't understand something, ask other students or just Google it.
- In the name of Talos, use your brain!

Act: Task 00

NAME

Mighty Wizard

DIRECTORY

`t00/`

BINARY

`mightyWizard`

LEGEND

Food is a category of consumable items that temporarily alter the Dragonborn's stats. This could mean replenishing or increasing one's maximum Health, Stamina or Magicka or changing other aspects such as magic resistance or bolstering stats. Of course, every wizard can identify what type of food is served to him, even if it is hidden behind the illusion.

DESCRIPTION

Create a program that has:

- `AbstractWizard` abstract class that deducts the food type from a food object. It contains `deductFoodType` member function which is only accessible by any derived class and inaccessible outside of class
- `FoodType` enum class that contains `ApplePie`, `HoneyNut`, `SweetRoll`, `PoisonedFood` and `Invalid`
- `FoodItem` base class and its 4 derived classes listed in `enum`, with the exception of `Invalid` which stands for invalid items
- `MightyWizard` class that inherits from `AbstractWizard`
- `checkFood` member function in `MightyWizard` class checks `FoodType` object and prints his reaction:
  - if the food is okay - prints `<food name>. Mmm, tasty!`
  - if it isn't - prints `<food name>. Ugh, not again!`

Remember, Mighty Wizards don't like poisoned food (does anyone?) and HATES apple pies. The solution output must be identical with CONSOLE OUTPUT while using the `main.cpp` from the SYNOPSIS.

SYNOPSIS

```
int main() {
    std::string name = "Elderic";
    MightyWizard wizard(name);

    FoodItem* piePtr = new ApplePie;
```

```
    FoodItem* rollPtr = new SweetRoll;
    FoodItem* nutPtr = new HoneyNut;
    FoodItem* poisonPtr = new PoisonedFood;

    auto& pieRef = *piePtr;
    auto& rollRef = *rollPtr;
    auto& nutRef = *nutPtr;
    auto& poisonRef = *poisonPtr;

    wizard.checkFood(piePtr);
    wizard.checkFood(rollPtr);
    wizard.checkFood(nutPtr);
    wizard.checkFood(poisonPtr);
    wizard.checkFood(nullptr);

    wizard.checkFood(pieRef);
    wizard.checkFood(rollRef);
    wizard.checkFood(nutRef);
    wizard.checkFood(poisonRef);

    delete piePtr;
    delete rollPtr;
    delete nutPtr;
    delete poisonPtr;

    return 0;
}
```

CONSOLE OUTPUT

```
>./mightyWizard | cat -e
Apple pie. Ugh, not again!$
SweetRoll. Mmm, tasty!$
Honey nut. Mmm, tasty!$
Poison. Ugh, not again!$
<wtf>. Ugh, not again!$
Apple pie. Ugh, not again!$
SweetRoll. Mmm, tasty!$
Honey nut. Mmm, tasty!$
Poison. Ugh, not again!$
>
```

SEE ALSO

Abstract class

Act: Task 01

NAME

Lockpicking

DIRECTORY

`t01/`

BINARY

`lockpicking`

LEGEND

The art of Lockpicking is used to open locked doors and containers faster and with fewer broken lockpicks.
What can you do with a simple chest? Of course, you can open it! And if it is locked? Try to lockpick it!

DESCRIPTION

Create a program that allows to simulate lockpicking of different locks by a player. The program uses `main.cpp` from the SYNOPSIS and produces the output listed in the CONSOLE OUTPUT. Let's start. Create

- `LockpickDifficulty` enum class that contains `None = 0`, `Novice = 25`, `Apprentice = 40`, `Adept = 60`, `Expert = 80` and `Master = 100`
- two interfaces:
  - `IOpenable` that contains `bool tryToOpen (LockpickDifficulty)` member function
  - `ILockable` that contains `LockpickDifficulty lockDifficulty()` and `bool isLocked()` member functions
- `Container` abstract class that has:
  - inheritance from `ILockable` and `IOpenable` interfaces
  - fields:
    - `m_isLocked` that indicates whether the container is locked or unlocked (unlocked by default)
    - `m_difficulty` that indicates the difficulty of lockpicking the container
  - constructor
  - `lockDifficulty` member function that returns `m_difficulty`
  - `isLocked` member function that returns `m_isLocked`
  - `tryToOpen` member function that returns:
    - `false` when `skill` is less than lockpicking difficulty
    - `true` when container isn't locked or if the `skill` is greater than or equal to
      the lockpicking difficulty, and sets the container to unlocked state

- `name` member function that returns name of the class

- `Barrel`, `Chest` and `Safe` derived classes from `Container` that contains:
  - constructor
  - `name` member function called the class name
- `Barrel` class objects always can be opened
- `PLayer` class that contains:
  - fields `m_lockpickSkill` which is `None` by default and `m_name`
  - constructor
  - `openContainer` member function where a player tries to lockpick container with his skill. It prints:
    * in case of fail - `<player name> skill is too low to open <container name>`
    * in case of success - `<player name> successfully opened <container name>`
  - `setLockpickSkill` member function sets the appropriate value of the parameter to the `m_lockpickSkill`

SYNOPSIS

```
/* Container.h */

class Container : public ILockable, public IOpenable {
 public:
    Container(bool isLocked, const LockpickDifficulty difficulty);

    LockpickDifficulty lockDifficulty() const;
    bool isLocked() const;
    bool tryToOpen(LockpickDifficulty skill);

    virtual std::string name() const = 0;

 private:
    bool m_isLocked{false};
    LockpickDifficulty m_difficulty;
};

/* Player.h */

class Player final {
 public:
    explicit Player(const std::string& name);

    void openContainer(Container* container);

    void setLockpickSkill(LockpickDifficulty skill);
```

```
 private:
    LockpickDifficulty m_lockpickSkill{LockpickDifficulty::None};
    std::string m_name;
};
```

```
/* main.cpp */

int main() {
    std::string playerName = "J'datharr";
    auto player = Player(playerName);

    auto barrel = new Barrel;
    auto chest = new Chest(true, LockpickDifficulty::Apprentice);
    auto safe = new Safe(true, LockpickDifficulty::Master);

    player.openContainer(barrel);
    player.openContainer(chest);
    player.openContainer(safe);

    player.setLockpickSkill(LockpickDifficulty::Adept);

    player.openContainer(chest);
    player.openContainer(safe);

    player.setLockpickSkill(LockpickDifficulty::Master);

    player.openContainer(safe);

    delete barrel;
    delete chest;
    delete safe;

    return 0;
}
```

CONSOLE OUTPUT

```
>./lockpicking | cat -e
J'datharr successfully opened Barrel$
J'datharr skill is too low to open Chest$
J'datharr skill is too low to open Safe$
J'datharr successfully opened Chest$
J'datharr skill is too low to open Safe$
J'datharr successfully opened Safe$
>
```

SEE ALSO

Interface Class
Interfaces in C++ (Abstract Classes)

Act: Task 02

NAME

Magic Spells

DIRECTORY

`t02/`

BINARY

`magicSpells`

LEGEND

The greatest of all the troubles that may threaten the Empire in the future, without a doubt, is associated with the presence of redguards in one of its areas. The Redhards have always had a bad relationship with the Imperial soldiers.

DESCRIPTION

It's time for some magic duels. Recreate a battle between a Redguard and an Imperial in `main.cpp` so the output of the program is identical with the output listed in the CONSOLE OUTPUT. Create the following:

- `ISpell` interface that contains:
  - function declaration `cast`
  - function declaration `getType`
  - overloaded `operator==` that compares the objects by their types using `getType`
- `SpellType` enum class that contains `Healing`, `Equilibrium`, `Flames`, `Freeze`, `Fireball`
- five derived classes using `SpellType` enum that implements `ISpell` interface:
  - `cast` member function has the following logic for classes:
    - `Healing` - the owner restores 10 health and loses 15 mana
    - `Equilibrium` - the owner restores 25 mana and loses 25 health
    - `Flames` - the owner loses 14 mana, and other loses 8 health
    - `Freeze` - the owner loses 30 mana, and other loses 20 health
    - `Fireball` - the owner loses 50 mana, and other loses 40 health

    Check if there are enough `mana` to cast a `spell`.
  - `getType` member function returns type of class using `SpellType` enum
- `Creature` class that contains:
  - fields: `m_name`, `m_health`, `m_mana` and `m_spells`
  - constructor:

    - takes `rvalue` reference parameter called `name` and initializes `m_name` with the given value
    - always initializes next class members as follows `m_health=100`, `m_mana=100`
    - prints `<m_name> was born!` to the standard output
  - destructor that clears `m_spells`
  - `learnSpell` member function tries to add `spell` to the `m_spells` and prints the respective message:
    - if success - prints `<creatureName> has learned <spellName> spell successfully!`
    - if failure - prints `<creatureName> already knows <spellName> spell!`
  - `castSpell` member function tries to cast `spell` on `creature` and prints the respective message:
    - `<spellName> spell is not learned by <creatureName>` if the spell isn't learned by a creature who casts it
    - `<creatureName> casted <spellName> spell on <creatureName>!` if the casting of spell on the creature is a success
    - `<creatureName> can't cast <spellName>!` if the casting of spell on the creature is a failure
  - `operator<<` overloaded operator that prints a message to the standard output - `<creatureName> : <creatureHealth> HP, <creatureMana> MP.`
- two derived classes from `Creature` class called `Imperial` and `Redguard`.
  Both of them override `sayPhrase` member function and print a phrase to the standard output:
  - `I am <name> from Redguards, wanna trade?` for Redguard
  - `I am <name>, Imperial soldier! Stop right here!` for Imperial

SYNOPSIS

```
/* Creature.h */

namespace Creatures {

class Creature {
 public:
    explicit Creature(const std::string& name);
    virtual ~Creature();

    void learnSpell(Spells::ISpell* spell);
    void castSpell(const Spells::SpellType type, Creature& creature);
    virtual void sayPhrase() const = 0;

    std::string getName() const;
```

```
    int getHealth() const;
    int getMana() const;

    void setHealth(int health);
    void setMana(int mana);

 private:
    std::string m_name;
    int m_health;
    int m_mana;
    std::set<Spells::ISpell*> m_spells;
};

}  // end namespace Creatures

std::ostream& operator<<(std::ostream& os, const Creatures::Creature& creature);

/* ISpell.h */

namespace Spells {

class ISpell {
 public:
    virtual ~ISpell() = default;

    virtual bool cast(Creatures::Creature& owner, Creatures::Creature& other) = 0;
    virtual SpellType getType() const = 0;
};

}  // end namespace Spells

bool operator==(Spells::ISpell& lhs, Spells::ISpell& rhs);
```

CONSOLE OUTPUT

```
>./magicSpells | cat -e
JoJo was born!$
I am JoJo from Redguards, wanna trade?$
Dio was born!$
I am Dio, Imperial soldier! Stop right here!$
JoJo : 100 HP, 100 MP.$
Dio : 100 HP, 100 MP.$
JoJo has learned healing spell successfully!$
JoJo has learned fireball spell successfully!$
Dio has learned equilibrium spell successfully!$
Dio already knows equilibrium spell!$
JoJo casted fireball spell on Dio!$
Dio casted equilibrium spell on JoJo!$
JoJo : 75 HP, 50 MP.$
Dio : 60 HP, 100 MP.$
flames spell is not learned by Dio.$
Dio has learned flames spell successfully!$
```

```
Dio has learned freeze spell successfully!$
JoJo casted healing spell on JoJo!$
Dio casted freeze spell on JoJo!$
JoJo : 65 HP, 35 MP.$
Dio : 60 HP, 70 MP.$
JoJo can't cast fireball!$
JoJo can't cast fireball!$
JoJo can't cast fireball!$
JoJo : 65 HP, 35 MP.$
Dio : 60 HP, 70 MP.$
>
```

SEE ALSO

Operator overloading

Share

PUBLISHING

Last but not least, the final stage of your work is to publish it. This allows you to share your challenges, solutions, and reflections with local and global audiences. During this stage, you will discover ways of getting external evaluation and feedback on your work. As a result, you will get the most out of the challenge, and get a better understanding of both your achievements and missteps.

To share your work, you can create:

- a text post, as a summary of your reflection
- charts, infographics or other ways to visualize your information
- a video, either of your work, or a reflection video
- an audio podcast. Record a story about your experience
- a photo report with a small post

Helpful tools:

- Canva - a good way to visualize your data
- QuickTime - an easy way to capture your screen, record video or audio

Examples of ways to share your experience:

- Facebook - create and share a post that will inspire your friends
- YouTube - upload an exciting video
- GitHub - share and describe your solution
- Telegraph - create a post that you can easily share on Telegram
- Instagram - share photos and stories from ucode. Don't forget to tag us :)

Share what you've learned and accomplished with your local community and the world. Use #ucode and #CBLWorld on social media.